Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №2.2

з дисципліни
«Інтелектуальні вбудовані системи»

на тему
«ДОСЛІДЖЕННЯ АЛГОРИТМУ ШВИДКОГО ПЕРЕТВОРЕННЯ ФУР'Є З ПРОРІДЖУВАННЯМ ВІДЛІКІВ СИГНАЛІВ У ЧАСІ»

Виконав:

студент групи ІП-84
Ковалишин Олег Юрійович
номер залікової книжки: 8410

Перевірив:

ас. кафедри ОТ
ас. Регіда П. Г.

Київ 2021

2. Теоретичні відомості

Швидкі алгоритми ПФ отримали назву схеми Кулі-Тьюкі. Всі ці алгоритми використовують регулярність самої процедури ДПФ і те, що будь-який складний поворотний коефіцієнт можна розкласти на прості комплексні коефіцієнти.

Алгоритм Кулі-Тьюкі:

$$F_x(p) = \underbrace{\sum_{k^*=0}^{\frac{N}{2}-1} X(2k^*) W_{\frac{N}{2}}^{pk^*}}_{F_{II}(p^*)} + \underbrace{W_N^p \sum_{k^*=0}^{\frac{N}{2}-1} X(2k^*+1) W_{\frac{N}{2}}^{pk^*}}_{F_I(p^*)}$$

Ми бачимо, що всі вирази можна розділити на 2 частини, які обчислюються паралельно. F(p*) - проміжний спектр, побудований на парних відліку. У цьому алгоритмі передбачається, щоб отримати спектр F(p) треба виконати 2 незалежних N/2 ШПФ.

$$1)\quad F_{II}(p^*) = \sum_{k^*=0}^{\frac{N}{2}-1} X(2k^*) W_{\frac{N}{2}}^{pk^*}$$

$$2)\quad F_I(p^*) = \sum_{k^*=0}^{\frac{N}{2}-1} X(2k^*+1) W_{\frac{N}{2}}^{pk^*}$$

$$F_\tau(p^*) = F_{II}(p^*) + W_N^{p^*} F_I(p^*)$$

3. Умови завдання

Варіант 10:

n = 14, ω гр = 1700, N = 64

4. Вихідний код

```
fun main() {
    plotDFT(14, 1700, 1024, true)
    plotFFT(14, 1700, 1024, true)
    plotO(14, 1700, 1000, 100_000, 1000, 5_000)
//    benchmarkFT(14, 1700, 10024)
}

import kotlin.system.measureTimeMillis

fun benchmarkFT(n: Int, wMax: Int, num: Int) {
    val s = Signal(n, wMax, num)
    val timeDFT = measureTimeMillis { s.dft() }
    println("DFT: $timeDFT msec")
    val timeFFT = measureTimeMillis { s.fft() }
    println("FFT: $timeFFT msec")
    println("DFT/FFT: ${timeDFT.toFloat() / timeFFT}")
}

import kotlin.math.*

data class Complex(val r: Double, val i: Double) {
    operator fun times(times: Float) = Complex(r * times, i * times)
    operator fun times(times: Double) = Complex(r * times, i * times)
    operator fun times(times: Complex) = Complex(r * times.r - i * times.i, i * times.r + r * times.i)
    operator fun plus(that: Complex) = Complex(this.r + that.r, this.i + that.i)
    operator fun plus(that: Double) = Complex(this.r + that, this.i)
    operator fun minus(that: Complex) = Complex(this.r - that.r, this.i - that.i)
    fun abs() = sqrt(r.pow(2) + i.pow(2))
}

operator fun Double.plus(that: Complex) = Complex(that.r + this, that.i)
operator fun Double.minus(that: Complex) = Complex(this - that.r, -that.i)
```

```
typealias W = (Int) -> Complex

fun wWithBase(n: Int): (Int) -> Complex {
    val arg = 2 * PI / n
    val cache = mutableMapOf<Int, Complex>()

    return { i: Int ->
        cache.getOrPut(i % n) {
            Complex(cos(arg * i), -sin(arg * i))
        }
    }
}

fun dft(values: List<Float>, w: W? = null): List<Complex> {
    val num = values.size
    val w = w ?: wWithBase(num)

    return List(num) { p ->
        var f = Complex(0.0, 0.0)
        for (k in 0 until num) f += w(p * k) * values[k]
        f
    }
}

fun Signal.dft(normed: Boolean = false): List<Double> {
    val f = dft(y).map { it.abs() / num }
    return if (normed) f.map { it * 2 }.dropLast(num / 2) else f
}

private fun fft(values: List<Float>): List<Complex> {
    val n = values.size
```

```
    val w = wWithBase(n)
    if (n <= 32) return dft(values)

    val half = n / 2
    val even = MutableList(n / 2) { 0f }
    val odd = MutableList(n / 2) { 0f }

    for (i in 0 until half) {
        even[i] = values[2 * i]
        odd[i] = values[2 * i + 1]
    }

    val xEven = fft(even)
    val xOdd = fft(odd)

    val f = MutableList(n) { Complex(0.0, 0.0) }
    for (p in 0 until half) {
        f[p] = xEven[p] + w(p) * xOdd[p]
        f[half + p] = xEven[p] - w(p) * xOdd[p]
    }

    return f
}

fun Signal.fft(normed: Boolean = false): List<Double> {
    val f = fft(y).map { it.abs() / num }
    return if (normed) f.map { it * 2 }.dropLast(num / 2) else f
}

import kscience.plotly.*
import kscience.plotly.models.XAnchor
import kscience.plotly.models.YAnchor
```

```
import kscience.plotly.palettes.Xkcd
import kotlin.system.measureTimeMillis

fun plotDFT(n: Int, wMax: Int, num: Int, normed: Boolean = false) {
    val s = Signal(n, wMax, num)
    val dft = s.dft(normed)

    Plot("w", "A").apply {
        addLine(dft.indices, dft, Xkcd.BLUE, "DFT")
    }.draw()
}

fun plotFFT(n: Int, wMax: Int, num: Int, normed: Boolean = false) {
    val s = Signal(n, wMax, num)
    val fft = s.fft(normed)

    Plot("w", "A").apply {
        addLine(fft.indices, fft, Xkcd.BLUE, "FFT")
    }.draw()
}

fun plotO(
    n: Int,
    wMax: Int,
    numMin: Int,
    numMax: Int,
    numStep: Int,
    maxTime: Int,
) {
    val dftTimes = mutableListOf<Long>()
    for (i in numMin..numMax step numStep) {
        val s = Signal(n, wMax, i)
```

```
        val time = measureTimeMillis { s.dft() }
        println("DFT FOR $i: $time")
        dftTimes.add(time)
        if (time >= maxTime) break
    }

    val fftTimes = mutableListOf<Long>()
    for (i in numMin..numMax step numStep) {
        val s = Signal(n, wMax, i)
        val time = measureTimeMillis { s.fft() }
        println("FFT FOR $i: $time")
        fftTimes.add(time)
        if (time >= maxTime) break
    }

    val x = fftTimes.indices.map { it * numStep + numMin }

    Plot("w", "A").apply {
        addLine(x, dftTimes, Xkcd.BLUE, "DFT")
        addLine(x, fftTimes, Xkcd.RED, "FFT")
    }.draw()
}

private class Plot(
    private val xAxis: String?,
    private val yAxis: String?,
) {
    private val lines = mutableListOf<Line>()

    fun addLine(x: Iterable<Number>, y: Iterable<Number>, color: String, name: String) {
        lines += Line(x, y, color, name)
    }
```

```
fun draw() {
    Plotly.page(mathJaxHeader, cdnPlotlyHeader) {
        plot {
            lines.forEach { line ->
                scatter {
                    x.set(line.x)
                    y.set(line.y)
                    line { color(line.color) }
                    name = line.name
                }
            }

            layout {
                height = 750
                width = 1000
                margin { l = 50; r = 20; b = 20; t = 50 }
                xaxis.title = xAxis
                yaxis.title = yAxis
                legend {
                    x = 0.97
                    y = 1
                    borderwidth = 1
                    font { size = 32 }
                    xanchor = XAnchor.right
                    yanchor = YAnchor.top
                }
            }
        }
    }.makeFile()
}
```

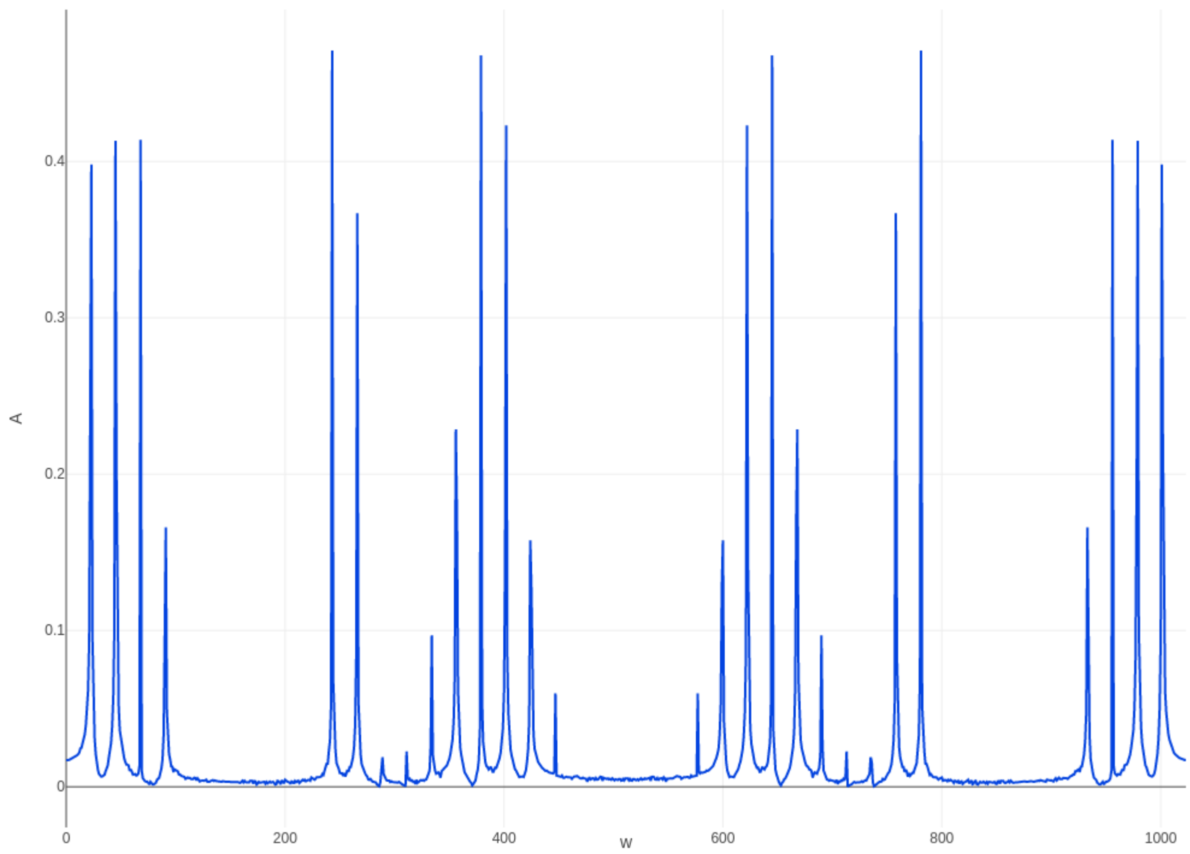
```
    private class Line(
        val x: Iterable<Number>,
        val y: Iterable<Number>,
        val color: String,
        val name: String,
    )
}

import java.util.*
import kotlin.math.min
import kotlin.math.pow
import kotlin.math.sin
import kotlin.math.sqrt

class Signal(val n: Int, val wMax: Int, val num: Int) {
    var values: Array<Float> = arrayOf()
        get() {
            if (field.size < num) generate()
            return field
        }
        private set

    val x: List<Int>
        get() = values.indices.toList()

    val y: List<Float>
        get() = values.toList()

    val m
        get() = values.average()

    val d
```

```
    get() = values.map { (it - m).pow(2) }.sum() / (num - 1)

private fun generate() {
    val random = Random()
    val signals = Array(num) { 0f }
    for (i in 1..n) {
        val a = random.nextFloat()
        val fi = random.nextFloat()
        val w = wMax.toFloat() * i / n
        for (t in 0 until num) {
            val s = a * sin(w * t + fi)
            signals[t] += s
        }
    }

    values = signals
}

infix fun tau(tau: Int): Signal {
    if (tau >= num) error("Invalid tau: $tau/$num")
    return Signal(n, wMax, num - tau).also {
        it.values = values.drop(tau).toTypedArray()
    }
}

fun correlation(that: Signal, normed: Boolean = false): Float {
    val n = min(this.num, that.num)
    var cov = 0f.toDouble()

    val x = this.values
    val mx = this.m
    val y = that.values
```

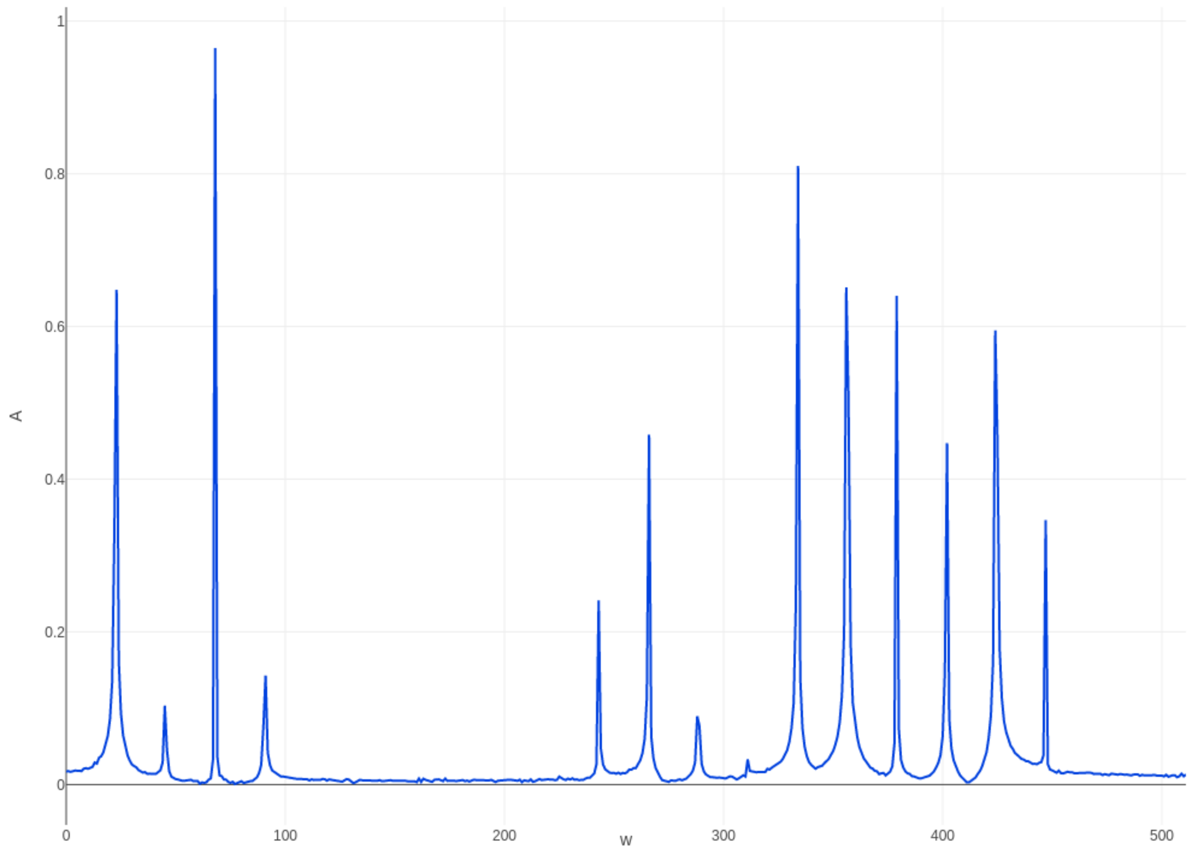
```
        val my = that.m

        for (i in 0 until n) cov += (x[i] - mx) * (y[i] - my)

        val dx = x.map { (it - mx).pow(2) }.sum()
        val dy = y.map { (it - my).pow(2) }.sum()

        val corr = if (normed) cov / sqrt(dx * dy) else cov / (n - 1)
        return corr.toFloat()
    }
}
```

5. Результати виконання програми

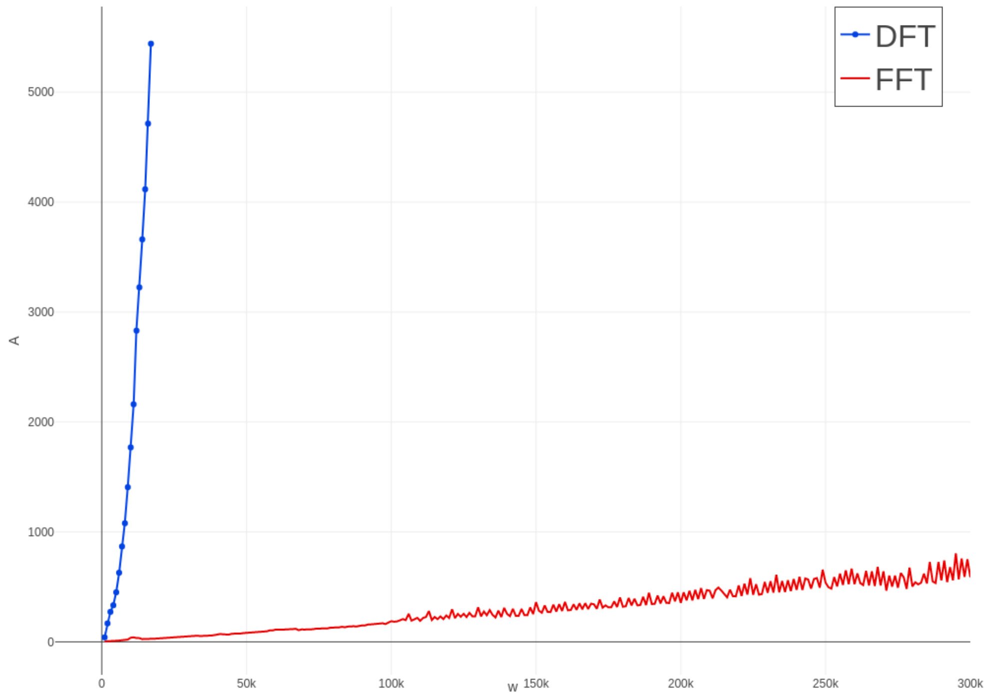
ШПФ, ненормовано:



ШПФ, нормовано:



Порівняння швидкодії БПФ (з табличним методом) та ШПФ:



6. Висновки щодо виконання лабораторної роботи.

У ході виконання лабораторної роботи проведено ознайомлення ознайомлення з принципами реалізації прискореного спектрального аналізу випадкових сигналів на основі алгоритму швидкого перетворення Фур'є, вивчення та дослідження особливостей даного алгоритму з використанням засобів моделювання і сучасних програмних оболонок.